[Print](javascript:window.print())

**Course Transcript**

Communicating with WebSockets and Utilizing Asynchronous Processing

**Communicating with HTML5 WebSockets**

[1. Course Introduction](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t54)

[2. Introducing the HTML5 WebSocket Standard](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t4)

[3. Web Communication and Current Limitations](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t8)

[4. WebSocket API](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t12)

[5. Send WebSocket Processes](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t16)

[6. Receive WebSocket Processes](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t20)

**Web Workers and Asynchronous Processing**

[1. Introducing HTML5 Web Workers](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t25)

[2. Using the Web Worker API](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t29)

[3. Message Passing with HTML5](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t33)

[4. Callback Implementation](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t37)

[5. Implementing Shared Web Workers](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t41)

[6. Improving Web Page Performance Using Web Workers](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t45)

**Practice: Web Workers and WebSockets**

[1. Exercise: Processing and Web Workers in HTML5](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#t50)

Course Introduction

Learning Objective

*After completing this topic, you should be able to*

* *start the course*

**1.**

*[Course title: Communicating with WebSockets and Utilizing Asynchronous Processing.]*Hi, I'm Jonathan Ken. In this course, William Shen *[IT Consultant.]*will cover the WebSocket API and web workers. He'll also provide in-depth coverage of the web worker processes.

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

Introducing the HTML5 WebSocket Standard

Learning Objective

*After completing this topic, you should be able to*

* *define the HTML5 WebSocket standard and how WebSockets work*

**1.**

*[Topic title: Introducing the HTML5 WebSocket Standard. The presenter is William Shen. A diagram is displayed in which three browsers are connected to the WebSocket Server via Internet, which is connected to the Application Servers.]*In this video, we will be introducing the HTML5 WebSocket standard. A WebSocket standard is a method for creating a connection between a client and the server. It provides method to send and receive full duplex messages. It uses the protocol specification WebSocket and WebSocket secure. WebSocket are bidirectional, full-duplex, persistent connection from the web browser to a server. Once a WebSocket connection is established, the connection stays open until the client or server decides to close this connection. With this open connection, the client or server can send messages at any given time to the other. This makes web programming entirely event-driven and not just user initiated, it is stateful.

As well at this time, a single running server application is aware of all connections, allowing you to communicate with any number of open connection at any given time. Here we have a diagram of parts of a WebSocket connection. *[The Diagram consists of the WebSocket Client, API Gateway, and the WebSocket Server.]*It first initialize the connection through the WebSocket client, sending an HTTP upgrade request to WebSocket server. The server returns a 101 response acknowledging the connection. The bidirectional channel is then established. The client and server engage in bidirectional messaging, which is the sending and receiving bidirectional messages. The server notifies the client about connection closed. And client response to the connection close message. Then, the connection is closed. This concludes our video on introduction to the HTML5 WebSocket standard.

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

Web Communication and Current Limitations

Learning Objective

*After completing this topic, you should be able to*

* *recognize various web browser communications and limitations*

**1.**

*[Topic title: Web Communication and Current Limitations. The presenter is William Shen.]*In this video, we will be talking about Web communication and current limitations that WebSocket has. Some of the limitation we see that browser may have is preconfigured and user configurable connection limits. Server load can be significantly impacted because of concurrent connections and there can be delay when opening and closing the WebSocket that must be handled.

Web communication is supported by the following browsers. Internet Explorer version 10 or later, Firefox version 11 or later, Google Chrome version 16 and later, Safari version 5.1 and later, Opera version 12.10 and later, iOS 6.0 and later, and Android version 4.4 and later. This concludes our brief discussion about Web communication and current limitations.

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

WebSocket API

Learning Objective

*After completing this topic, you should be able to*

* *recognize how to set up a WebSocket connection*

**1.**

*[Topic title: WebSocket API. The presenter is William Shen. The WebSite1 project is open in the Microsoft Visual Studio window. The open.html tabbed page is open. The Solution Explorer is open in the navigation pane.]*In this video, I will demonstrate how to open and close a WebSocket connection. Looking at our screen here, *[The entire code is mentioned at the end of the transcript under the heading,"open.html".]*you'll find we have two links on our web page. One is to open the connection, the other is to close it. *[He points at <a href="javascript:WebSocketOpen()">Open WebSocket</a> and <a href="javascript:WebSocketClose()">Close WebSocket</a>.]*In our JavaScript here we'll specify a variable, ws, it stands for WebSocket. *[He types var ws under <script type="text/javascript">.]*And in our WebSocket open function. We will say if, and the first thing we need to do is to check if the browser supports WebSocket. So if WebSocket in window, then ws equals to new WebSocket, and the string is the WebSocket address. For our address here, I have ws://localhost. This address leads to a WebSocket resource I have created particularly for this tutorial. So the location is our localhost port 8080/echo/echoWs, this will echo back any request from the client. Then we need to set up some listening command. So we have ws.onopen = function(), then alert("Web Socket is connected");. We will do the same with close. So ws.onclose = function(), and then alert(" Web Socket is closed");.

We then need to specify an else case, in the event that WebSocket is not supported. So under else, we'll have here alert("Web Socket is NOT supported by your Browser!);. So that's it for open function. Our close function will simply be ws.close();. Let's give this a test run. *[He right-clicks the open.html file under webSocket folder in the Solution Explorer. He selects the View in Browser(Google Chrome) option. The localhost:51430/webSocket/open.html web page opens.]*So on our browser here, you can see we have two links available. One is to Open WebSocket and one is Close WebSocket. Let's open our inspection window and select the Network tab. If I am to click on Open we can see a message prompt that says Web Socket is open. And notice how the status is currently pending. Once we click on OK, the server responds with status 101. This means we are now connected at this point. If we choose to click on Close, we will receive a message indicating the Web Socket is now closed. This concludes our video on opening and closing a WebSocket connection.

**"open.html"**  
<!DOCTYPE HTML>  
<html>  
<head>  
<style>  
a {  
display: block;  
}  
</style>  
<script type="text/javascript">  
function WebSocketOpen() {  
}  
function WebSocketClose() {  
}  
</script>  
</head>  
<body>  
<a href="javascript:WebSocketOpen()">Open WebSocket</a>  
<a href="javascript:WebSocketClose()">Close WebSocket</a>  
</body>  
</html>

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

Send WebSocket Processes

Learning Objective

*After completing this topic, you should be able to*

* *recognize how to use WebSocket to send messages to a web server*

**1.**

*[Topic title: Send WebSocket Processes. The presenter is William Shen. The send.html tabbed page is open.]*In this video, we will demonstrate how to send data to a server using WebSocket. Here we have our page set up with a link to open the WebSocket connection, *[He points at <a href="javascript:WebSocketOpen()">Open WebSocket</a>.]*and a link to close WebSocket connection. *[He points at <a href="javascript:WebSocketClose()">Close WebSocket</a>.]*There's also a third link to trigger the send function that will send the message to the server. *[He points at <a href="javascript:WebSocketSend()">Send message</a>.]*Here within our WebSocketSend function, we will do ws.send and we will put in the message within the parameter. For this message, I will set Hello World with four exclamation marks. This is all we need to do to send message. Let's open this page up in the browser. On the page here we can see there is three links displayed. Go into our inspection window, *[He right-clicks the window and selects the Inspect option.]*select Console to see the output. If I click on Open WebSocket, we will see in our console the Web Socket is connected, to indicate the connection established. If I click on send message, which I'm going to click three times, to send the message Hello World to the server three times.

Then navigate to our GlassFish Server log. *[The GlassFish Server tabbed page opens in the echo - NetBeans IDE 8.1 window.]*This here is a WebSocket server hosted on the GlassFish Server. As we can see on our log, it prints out the message every time it receives a message. You can see the message has appeared three times. *[He points at Info: Message received: Hello World!!!.]*Going back to our webpage, we can still close the connection *[He clicks the Close WebSocket link.]*and then see the WebSocket is closed on our console. If we choose to send the message now, *[He clicks the Send message link.]*we will see an error message indicating WebSocket is already in closing or closed state. This concludes our video on how to send data to a server using WebSocket.

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

Receive WebSocket Processes

Learning Objective

*After completing this topic, you should be able to*

* *recognize how to receive messages from a web server via a WebSocket*

**1.**

*[Topic title: Receive WebSocket Processes. The presenter is William Shen. The receive.html tabbed page is open.]*In this video, we will demonstrate how to receive WebSocket messages. In our previous video, we have successfully sent out Hello World message to the server. Now we are going to try and receive a response for our message of Hello World. To do so, we will say ws.onmessage. *[The reference code is mentioned at the end of the transcript under the heading, "receive.html".]*This function will trigger when we receive a message from the server. So we have onmessage = function, and we will create a variable called var received\_msg. And we should remember that we need to put in the parameter event. So received\_msg = evt.data. And we are going to log("message is received + received\_msg). Let's take a look at the preview. Here is our page with three links to select. Let's open up our inspection first and select Console. And we will open the connection and send a message. As you can see inside the Console, Web Socket is connected. And shows message is received, and the message is Hello World!!!!

Then we can go on and close our connection. And once we do, we will no longer be able to send and receive messages. Another way we can receive message is by the server sending message directly. The purpose of directional communication is that we can send a server request and the server can also initiate a message towards the client. So here, instead of connecting to our echo WebSocket, we will connect to the timer WebSocket. *[He alters the code from ws = new WebSocket("ws: //localhost:8080/echo/echoWs") to ws = new WebSocket("ws://localhost:8080/echo/timerWs").]*This is another WebSocket I created that will send the current time information every second. Let's take a look at how this will work. *[He right-clicks the receive.html file in the Solution Explorer and selects View in Browser(Google Chrome) option.]*So we will open our Inspection window and select Console. Then click on Open WebSocket to start our communication. You will see that we are receiving a local time message every second. *[He points at message is received .... Local time in the Console.]*If we click on Close connection, *[He clicks at the Close WebSocket link.]*then the message receiving will stop. This concludes our video on how to receive WebSocket message on the client side.

**"receive.html"**  
<script type="text/javascript">  
var ws;  
function WebSocketOpen(){  
if ("WebSocket" in window){  
ws = new WebSocket("ws: //localhost:8080/echo/echoWs");  
ws.onopen = function () {  
console.log("Web Socket is connected");  
}  
ws.onmessage = function (evt) {  
var received msg = evt.data;  
console.log("message is received…." + received msg);  
}  
ws.onclose = function (){  
console.log("Web Socket is closed");  
}  
} else {  
console.log("Web Socket NOT supported by your Browser!");  
}  
}  
function WebSocketSend() {  
ws.send("Hello World!!!!");  
}  
function WebSocketClose() {  
ws.close();  
}  
</script>

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

Introducing HTML5 Web Workers

Learning Objective

*After completing this topic, you should be able to*

* *define HTML5 Web Workers and their purpose*

**1.**

*[Topic title: Introducing HTML5 Web Workers. The presenter is William Shen.]*In this video, we will be introducing HTML5 web workers. First, what is a web worker? A web worker is like a thread, it allows a script file or JavaScript code to run in the background, usually in parallel. This can help keep a site feeling responsive during large processing tasks. On this slide, you can see we have an example of a normal Java process flow, versus a process flow using web workers. *[In the Normal JavaScript Process Flow, Step 1, Step 2, and Step 3 in series produce a result. In Process Flow Using Web Workers, Step 1 is connected in parallel to Step 2 and Step 3, which in turn produce a result.]*The result is using web worker to obtain a result in a more efficient manner.

Now, how are web workers different from real threads? *[A diagram is displayed, which connects the Web Worker to the PORT, which is connected to another Port via a MESSAGING CHANNEL. The PORT is connected to the Page via postmessage. The Web Worker consists of Worker.js and the Page consists of onmessage.]*First, they do not support most threading features, like semaphores or mutexes. It does not have access to resources outside of its own code, and data must be passed into and retrieved from the web worker via messages. And when should a web worker be used? Web workers should be used when a task or process can be done in parallel, or when page performance may be affected by slow input/output operation, or large calculations. There are two types of web workers, one is dedicated, which is the limited to the page that created it, and closing the page terminates the web worker. The other is shared, which is a page within the same web application can communicate with it. Leaving the web app terminates the web worker. This concludes our video on introduction to HTML5 web workers.

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

Using the Web Worker API

Learning Objective

*After completing this topic, you should be able to*

* *recognize how to create a Web Worker and control a Web Worker from the main page*

**1.**

*[Topic title: Using the Web Worker API. The presenter is William Shen. The connect.html tabbed page is open.]*In this video, we will demonstrate how to create a web worker in our main page. *[The entire code is mentioned at the end of the transcript under the heading, "connect.html".]*In our main page here, you can see we have two buttons, one to start web worker and one to stop web worker. Let's work on start web worker first. Before we do, let's create a variable called w1, I'll set the script. The first thing we want to do with web worker is to check if the web browser supports it. We can use the type of function. So type of worker does not equal to undefined. *[He types if (typeof (Worker) !== "undefined") { after function startWorker1() {.]*Because if it is equal to undefined, then we need to say alert. I will give a message, Sorry your browser doesn't support Web Worker. *[He types } else {, alert(Sorry your browser doesn't support Web Worker").]*If the web browser supports it, we will check if the type of w1.

So we will check if the workers already initialized for w1. If w1 is equal to undefined, meaning it is not created yet, we will say w1 equals to new worker and we will use demo\_workers.js. *[He types, code starts: if (typeof (w1) == "undefined") {, w1 = new Worker("demo\_workers.js"). Code ends.]*Notice on the side here, under our demo directory, we have demo\_workers.js defined in here. *[He points at the demo\_worker.js tabbed page.]*This is going to be our customized js file, that defines our web worker. *[He switches to the connect.html tabbed page.]*More detail on this in the next video. With what we have here now, we completed creating the web worker and connecting to it. Let's work on the stop worker now. So under function we will have w1.terminate. And w1 equals to undefined. There won't be much to demonstrate what we did. Simply when you press the start worker button, you will start the connection. And if you click on the stop worker, you will terminate the connection. This concludes our video on creating a web worker from the main page.

**"connect.html"**  
<!DOCTYPE html>  
<html>  
<body>  
<p>Time: <output id="result1"></output></p>  
<button onclick="startWorker1()">Start Worker1</button>  
<button onclick="stopWorker1()">Stop Worker1</button>  
<script>  
function startWorker1() {  
}  
function stopWorker1() {  
}  
</script>  
</body>  
</html>

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

Message Passing with HTML5

Learning Objective

*After completing this topic, you should be able to*

* *recognize the process of message passing in HTML5 with Web Workers*

**1.**

*[Topic title: Message Passing with HTML5. The presenter is William Shen. The send.html tabbed page is open.]*In this video, we will demonstrate how to send a message to the webWorker. Continuing from the previous video, we have two buttons on our page. One is startWorker, the other is stopWorker. So let's create a third button. And we will call this button Send Message. And for the Send Message we'll call the send message function, called sendMessage1. *[He types <button onclick="sendMessage()">Send Message</button>.]*We'll create a function here, sendMessage1. Under this function, we will simply put in w1.postMessage. We can post any message here, and it will get passed to our webWorker. *[He types function sendMessage1() {, w1.postMessage(10000).]*We will send the number 10000 to our webWorker. We will be using this number in the moment. Now we have shown you how to send message from the main page. Let's take a look at how the webWorker deals with this message. Let's open up our webWorker folder under our Solution Explorer on the right hand side. And we will look for demo\_workers.js. *[He double-clicks demo\_workers.js file and the demo\_workers.js file opens. It is divided into three sections, which are WebSite1, <global>, and sleep.]*As you can see, currently there is not much inside. There is just a helper function that will sleep for an x number of milliseconds. What we will do here is we will addEventListener in here.

And we will grab the message from the main page and perform this function. *[He types addEventListener("message", function (evt) { in row 1 under in the WebSite1 section.]*We will do a time counter so we have variable date = new Date();. And variable currentDate = null, then we have do, CurrentDate = new Date();. And here's the catch. The webWorker can also send message back to the main page using the postMessage function. So we will send back currentDate to the main page. So to summarize this function,once received message from the main page, this webWorker will iterate through the currentDate and continuously sending them every second. So we will see a time counter every second. While currentDate - date less than event data. So the event data is the number we passed to the webWorker, which was 10000. Which means this function will run for ten seconds in realtime, and then stop. Lastly we have false. The last bullion is to indicate whether it uses capture. Now that we have our webWorker set up, we will demonstrate how the main page will handle the message in the next video. This concludes our video on how to send a message to the webWorker.

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

Callback Implementation

Learning Objective

*After completing this topic, you should be able to*

* *implement a callback by using an anonymous function*

**1.**

*[Topic title: Callback Implementation. The presenter is William Shen. The receive.html tabbed page is open.]*In this video, we will demonstrate how to write callback function to receive messages. So here on our main page, we have our webWorker set up. It has been created and we're able to send a message to the webWorker. After we create the webWorker, we're going to create the callback function. So we have w1.onmessage = the function (evt). And we're going to say document.getElementById("result1").innerHTML = evt.data;. This is all we need to do.

Now we have a fully working webWorker in main page. *[The reference code is mentioned at the end of the transcript under the heading, "receive.html".]*Let's take a look at it. So we're opening this page in Google Chrome. *[It contains three buttons, which are Start Worker1, Send Message, and Stop Worker1.]*And we're going to Start Worker and Send Message. Once we click on Send Message, we'll receive a time update every second. And this time counter will count for ten seconds. So if we send again, it will start counting again. Once we click on Stop Worker, it will stop immediately. This concludes our video on how to write callback function to receive messages.

**"receive.html"**  
<!DOCTYPE html>  
<html>  
<body>  
<p>Time: Output id="result1"></output></p>  
<button onclick="startWorker1()">Start Worker1</button>  
<button onclick="sendMessage1()">Send Message</button>  
<button onclick="stopWorker1()">Stop Worker1</button>  
<script>  
var w1;  
function startWorker1() {  
if (typeof (Worker) !=="undefined") {  
if (typeof (w1) == "undefined") {  
w1 = new Worker("demo\_workers.js");  
}  
w1.onmessage = function (evt) {  
document.getElementById("result1").innerHTML = evt.data;  
}  
} else {  
alert("Sorry your browser doesn't support Web Worker")  
}  
}  
function sendMessage1() {  
w1.postMessage(10000);  
}  
function stopworker1() {  
w1.terminate();  
w1 = undefined;  
}  
</script>  
</body>  
</html>

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

Implementing Shared Web Workers

Learning Objective

*After completing this topic, you should be able to*

* *recognize code processing with HTML5 Web Workers*

**1.**

*[Topic title: Implementing Shared Web Workers. The presenter is William Shen. The sharedWorker.html tabbed page is open. The reference code is mentioned at the end of the transcript under the heading, " "sharedWorker.html".]*In this video, we would demonstrate how to create a Shared Worker. We have our first Worker1 as a non shared worker, and we will copy them to create Worker2. So, it has a start worker button, a send button and a stop worker button To create our shared worker under the function startWorker2, we need to first check if the type of shared worker does not equal to undefined. Else, we will create an alert. For our alert message we will say, Sorry your browser doesn't support Shared Web Worker. In the case the web browser supports shared worker, we will say if (typeof (w2) == "undefined"), then we will have w2 = new SharedWorker. And we will put the JavaScript file in here, which we'll fill in later. And for this one it's called the shared\_demo\_workers.js. So our listener will be on the port for shared workers. Each of the shared worker has many different ports. You will need to bind to one of these ports to be able to see the message from the shared worker. So here we have w2.port.addEventListener("message", function(evt). And we're going to do document.getElementById("result2").innerHTML = evt.data; and then false. Then we will call w2 port starts. This will start our shared worker. For our send function, it is similar to our non shared function, but we will be needing to use port to post our message. So we will have w2 port post message. And I'll put in 10000. So it will perform similar function, and create time content for the shared worker as well. To close it, we will say w2.port.close() and w2 = undefined. This is our main page. *[The reference code is mentioned at the end of the transcript under the heading, "sharedWorker.html (updated file)".]*

Now scroll to our shared worker page, and create our shared worker. *[He switches to shared\_demo\_workers.js tabbed page.]*So we first need to define our array of ports. Then we will define an addEventListener. And we need to make it so it listens to connect. Once the main page tries to connect to the shared worker, we will do a var port = evt.ports. And we will take the first one in the array. So we are grabbing the port that's sent by the connection. And we need to add the port to our port array. So we have ports.push(port). Then for our ports, we will addEventListener on message function. And we will do another event when we receive a message. Inside it will be similar to what we have done before. We will take the current date and define a variable called currentDate, which we will start implementing equals to null. Then do currentDate = new Date();. Here is the different part. Instead of taking our port to perform post message, we're going to take every port on our port array and post message to all our tabs. So we have for (i = 0; i < ports.length; i++). Then we will have ports[i].postMessage, and the message will be currentDate. Then I am going to sleep for one second. And I'm going to do this while (currentDate - date < evt.data). And we end with false. Lastly, we need to call port.start() and we want to make sure this is set to false. This is all we have to do for our shared worker. *[The reference code is mentioned at the end of the transcript under the heading, "shared\_demo\_workers.js".]*

Let's give it a try to see how this work. *[He switches to the sharedWorker.html tabbed page and opens the sharedWorker.html page in the browser.]*So we have two sets of button, the first set is the non shared workers, and the second set is shared. *[He points at Start Worker1, Send Message, and the Stop Worker1 buttons for non shared workers and he points at Start Worker2, Send Message, and the Stop Worker2 buttons for shared workers.]*To see this working, we will open another tab. And for both of them we will Start Worker2. Only on the second tab, I am going to click on Send Message. You will see the message being displayed, and time is ticking. We navigate to our first tab, and you will see the message being displayed. So we've sent message to our shared worker. And the shared worker has sent the message to all the port that's connected to it. And the ticking should stop in ten seconds. You can stop the shared worker on the second tab. And the first tab was to continue. This concludes our video on how to create shared workers.

**"sharedWorker.html"**  
<!DOCTYPE html>  
<html>  
<body>  
<p>Time: <output id="result1"></output></p>  
<button onclick="startWorker1()">Start Worker1</button>  
<button onclick="sendMessage1()">Send Message</button>  
<button onclick="stopWorker1()">Stop Worker1</button>  
<p>Time: <output id="result2"></output></p>  
<button onclick="startWorker2()">Start Worker2</button>  
<button onclick="sendMessage2()">Send Message</button>  
<button onclick="stopWorker2()">Stop Worker2</button>  
<script>…</script>  
<script>  
var w2;  
function startWorker2() {  
}  
function sendMessage2() {  
}  
function stopWorker2() {  
}  
</script>  
</body>  
</html>

**"sharedWorker.html (updated file)"**  
<!DOCTYPE html>  
<html>  
<body>  
<p>Time: <output id="result1"></output></p>  
<button onclick="startWorker1()">Start Worker1</button>  
<button onclick="sendMessage1()">Send Message</button>  
<button onclick="stopWorker1()">Stop Worker1</button>  
<p>Time: <output id="result2"></output></p>  
<button onclick="startWorker2()">Start Worker2</button>  
<button onclick="sendMessage2()">Send Message</button>  
<button onclick="stopWorker2()">Stop Worker2</button>  
<script>…</script>  
<script>  
var w2;  
function startWorker2() {  
if (typeof (SharedWorker) !== "undefined") {  
if (typeof (w2) == "undefined")  
w2 = new SharedWorker("shared\_demo\_workers.js");  
w2.port.addEventListener("message", function (evt) {  
document.getElementById("result2").innerHTML = evt.data;  
}, false);  
w2.port.start();  
}  
} else {  
alert("Sorry your browser doesn't support Shared Web worker");  
}  
}

function sendMessage2() {  
w2.port.postMessage(10000);  
}  
function stopWorker2() {  
w2.port.close();  
w2 = undefined;  
}  
</script>

**"shared\_demo\_workers.js"**  
var ports = [];  
addEventListener("connect", function (evt) {  
var port = evt.ports[0];  
ports . push(port);  
port.addEventListener("message", function (evt) {  
var date = new Date();  
var currentDate = null;  
do {  
currentDate = new Date();  
for (i = 0; i < ports.length; i++) {  
ports[i].postMessage(currentDate);  
}  
sleep(1000);  
} while (currentDate - date < evt.data);  
}, false);  
port.start();  
}, false);  
function sleep(ms)…

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

Improving Web Page Performance Using Web Workers

Learning Objective

*After completing this topic, you should be able to*

* *recognize the benefits of Web Workers to improve performance*

**1.**

*[Topic title: Improving Web Page Performance Using Web Workers. The presenter is William Shen. The performance.html tabbed page is open.]*In this video, we will demonstrate how web worker may affect performances. Here in our main page, we have two buttons. *[The reference code is mentioned at the end of the transcript under the heading, "performance.html".]*One is going to start our regular process, the second will start our web worker process. So let's take a look at our regular process. It is basically a loop of five processes. Each is going to take about one second to run. After the process run, the console will print out text, done. So it is pretty straightforward. Then, let's take a look at our web worker processes. It also has a loop of file process, but each of these process starts a web worker that is inside the busy\_workers.js. When we start web worker, we will post message to the web worker and it will respond. When it responds, we will log done, here. Let's look at our busy\_workers.js. *[The reference code is mentioned at the end of the transcript under the heading, "busy\_workers.js".]*It is exactly the same process as we had before. So it sleeps for one second, and you post the message back. *[He switches to the performance.html tabbed page.]*

So give this a try, and we will see how it impacts performance. Let's do a test run. In our browser, we start by opening our console. Within the console, let's do the regular process first. When we click on Regular Process, *[He clicks the Start Regular Process button.]*it takes about one second for each of the process to be done. So this took approximately five seconds in total. Now let's start the Web Worker. *[He clicks the Start Web Worker button.]*You will notice the process have been completed. You will find the processor out of numerical order, as the process are asynchronous. That means if I run that again, it could be another random order of processes. But this process will take one-fifth of the time regular task takes, as all threads are performing at the same time. This concludes our demo on how web worker can make a difference on your webpage performance.

**"performance.html"**  
<meta charset="utf-8" />  
<title></title>  
</head>  
<body>  
<button onclick="startRegular()">Start Regular Process</button>  
<p>Regular Task: <output id="rResult"></output></p>  
<button onclick="startWebWorker()">Start Web Worker</button>  
<p>Webworker Task: <output id="wResult"></output></p>  
<script>  
function startWebWorker() {  
for (c = 1; c <= 5; c++) {  
var worker = new Worker("busy\_workers.js");  
worker.onmessage = function (evt) {  
console.log('done ' + evt.data);  
document.getElementById("wResult").innerHTML = 'done';  
}  
worker.postMessage(c);  
}  
}  
function startRegular() {

for (c=1; c <= 5; c++) {  
sleep(1000);  
console.log('done ' + c);  
}  
document.getElementById('rResult').innerHTML = 'done';  
};  
function sleep(ms)…  
</script>  
</body>

**"busy\_workers.js"**  
addEventListener("message", function (evt) {  
sleep(1000);  
postMessage(evt.data);  
});  
function sleep(ms)…

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

Exercise: Processing and Web Workers in HTML5

Learning Objective

*After completing this topic, you should be able to*

* *recognize the various WebSockets and methods with HTML5*

**1.**

*[Topic title: Exercise: Processing and Web Workers in HTML5. The presenter is William Shen.]*In this exercise, you will create a WebSocket connection, send a message to a server, retrieve a response from a server. Describe how Web Worker can improve the performance of a web page. Create a simple Web Worker that adds all the number from 1 to 100 million. And split the task across multiple Web Worker in order to calculate the answer. Now pause the video and perform the exercise. *[He switches to the Microsoft Visual Studio window, in which lesson8-1.html tabbed page is open.]*Let's start with the web socket exercise. *[The reference code is mentioned at the end of the transcript under the heading, "lesson8-1.html".]*We have three different anchors on our main page. One is going to trigger WebSocketOpen. The next one is going to trigger WebSocketSend. And the last one is WebSocketClose. What we're going to do here in the JavaScript is we will specify a variable WebSocket. And in the WebSocketOpen, we're going to check if websocket, in window. This checks for web browser support of WebSocket. Else, if it does not support, we're going to create an alert. And say your browser does not support web socket. In the case, we have web socket support, we will have ws = new WebSocket. And we're going to connect it to the local host. This is our local web socket end port. So we have port 8080/echo/echoWs. Then we're going to have onopen = function(). We are going to create an alert as well, and this will say Web Socket is connected. Then, ws.onmessage.

So on receive a message, we want to run function with the event. And we're simply going to alert. Message is received. And we'll print out this message. Plus event data. Then next we will have ws.onclose. When the connection is closed, we want to run function alert. That indicates connection is closed. That's it for our WebSocketOpen() scripts. Now we're going to send a message using the WebSocketSend() function. So here we have ws.send. And we will send Hello World. Lastly, for WebSocketClose, we will simply have ws.close. And we'll correct a spelling error here. *[The reference code is mentioned at the end of the transcript under the heading, "lesson8-1.html (updated file)".]*Let's give this a try in the browser. Once our page loads, you will see three links. *[The links are: Open WebSocket, Send message, and Close WebSocket.]*Let's click on the Open WebSocket. You'll receive a response indicating that WebSocket is connected. Click on Send and you'll receive an echo back message saying, message is received Hello World. And upon pressing Close Connection, you will say connection closed. This concludes our exercise on WebSocket.

Let's do the Web Worker exercise. *[The calculator\_workers.js tabbed page opens.]*We start this exercise by creating a JavaScript file containing the Web Worker script. We implement it by adding the eventListener. Because we want to listen for input from the main page. So we listen for message and it will trigger the function with these events. Inside of it, we will say the event is going to carry an array of two values. So we have array = evt.data. And we will specify a variable called total, That's equal to 0. Now we will give it a for loop, so we will have i = array[0], which is the first number in the array. i less or to equal array[1]. That's the second number in the array. Then i++. Next, we have total += i. Then we postMessage(total). So you will take an array of two numbers and compute the sum of all the number in between inclusively. And this is it for our web worker. *[The reference code is mentioned at the end of the transcript under the heading, "calculator\_workers.js".]*Coming back to our main page. *[He switches to the lesson8-2.html tabbed page.]*We have a button that starts our web worker. It will trigger startWebWorker. So what we will do here is we start by setting a variable called division. And this will control the number of thread running asynchronously. So we will set the division to 1 first. And we will set the numbersToAdd, equals to 100 million. Which is nine digits long. Then we will set var total, Equal to 0. And we will create a variable called completed to keep track of the number of web worker completed. We'll set this to 0 to start with.

Let's create a for loop. So we have i = 0. i is less than division, i++ less than set the starting number to numberToAdd / division \* i + 1. So that is the starting number. And then, let's set the end number = numbersToAdd / division \* (i + 1). So that's our end numberToAdd. Lastly, we'll create our worker = new Worker, pointing to (“calculator\_workers.js”). And worker.onmessage, so when the web worker respond, we will create a function(evt). And inside this function, we will set total += evt.data. So the total will add the web worker result on top of it. And we will implement, completed++, and if (completed == division) so if all the web worker has completed, we are going to set document.getElementById("wResult").innerHTML = total. After the worker gets initialized, we're going to call worker.postMessage, and we will put in array with start to end.

For the purpose of this exercise, we will also place a timer on here as well. So variable startTime = new Date and variable endTime = null. When all the web worker has completed, we will set the endTime equal to new Date. And document.getElementById("wResultTime").innerHTML = (endTime-startTime). So let's give this a try. *[The reference code is mentioned at the end of the transcript under the heading, "lesson8-2.html tabbed file.]*Currently, we have division = 1. So only one thread will be running. So let's Start Web Worker. *[He opens lesson8-2.html page in the web browser and clicks the Start Web Worker button.]*It took about 390 milliseconds. If we do it a couple of more times, we will find the time indicates hovering about 300 milliseconds. Now, we will set the division to 10. *[He switches to the lesson8-2.html page and alters var division = 1 to var division = 10.]*Which will start ten web workers. Each of them will take a portion between 1 and 100 million to perform the addition. Now we Start Web Worker again. This time, you will notice an improvement on the time, which is around 180 milliseconds. As we now have multiple thread performing the task at the same time. Though we set the division to 10, as more web worker create more overhead. The performance boost does scale in proportion, instead of one-tenth of the time. This concludes our exercise for Web Workers.

**"lesson8-1.html"**  
<!DOCTYPE HTML>  
<html>  
<head>  
<style>  
a {  
display: block;  
}  
</style>  
<script type="text/javascript">  
function WebSocketOpen() {  
}  
function WebSocketSend() {  
}  
function WebSocketClose() {  
}  
</script>  
</head>  
<body>  
<a href="javascript:WebSocketOpen()">Open WebSocket</a>  
<a href="javascript:WebSocketSend()">Send message</a>  
<a href="javascript:WebSocketClose()">Close WebSocket</a>  
</body>  
</html>

**"lesson8-1.html (updated file)"**  
<!DOCTYPE HTML>  
<html>  
<head>  
<style>  
a {  
display: block;  
}  
</style>  
<script type="text/javascript">  
var ws;  
function WebSocketOpen() {  
if ("WebSocket" in window) {  
ws = new WebSocket("ws://localhost:8080/echo/echoWs");

ws.onopen = function () {  
alert('Web Socket is connected');  
};  
ws.onmessage = function (evt) {  
alert('Message is received.... ' + evt.data);  
};  
ws.onclose = function () {  
alert('Connection is closed');  
}

} else {  
alert('your browser does not support web socket');  
}  
}  
function WebSocketSend() {  
ws.send("Hello World");  
}  
function WebSocketClose() {  
ws.close();  
}  
</script>  
</head>  
<body>  
<a href="javascript:WebSocketOpen()">Open WebSocket</a>  
<a href="javascript:WebSocketSend()">Send message</a>  
<a href="javascript:WebSocketClose()">Close WebSocket</a>  
</body>  
</html>

**"calculator\_workers.js"**  
addEventListener("message", function (evt) {  
var array = evt.data;  
var total = 0;  
for (i = array[0]; i <= array[1]; i++) {  
total += i;  
}  
postMessage(total);  
});

**"lesson8-2.html"**  
</head>  
<body>  
<button onclick="startWebWorker()">Start Web Worker</button>  
<p>Webworker Task: <output id="wResult"></output></p>  
<p>Time: <output id="wResultTime"></output></p>  
<script>  
function startWebWorker() {  
var startTime = new Date();  
var endTime = null;  
var division = 1;  
var numbersToAdd = 100000000;  
var total = 0;  
var completed = 0;  
for (i = 0; i < division; i++) {  
var start = numbersToAdd / division \* i+1;  
var end = numbersToAdd / division \* (i+1);  
var worker= new worker("calculator\_workers.js");  
worker.onmessage = function (evt) {  
total += evt.data;  
completed++;  
if (completed == division) {  
document.getElementById("wResult").innerHTML = total;  
endTime = new Date();  
document.getElementById("wResultTime").innerHTML = (endTime – startTime);  
}  
}  
worker.postMessage([start, end]);  
}  
}  
</script>  
</body>  
</html>

[Back to top](https://library.skillport.com/courseware/Content/cca/wd_phjc_a08_it_enus/output/html/course_transcript.html#top)

© 2018 Skillsoft Ireland Limited